Floating-Point IEEE Filter for TurboLinux* on the Intel® Itanium™ Architecture

Application Note

March 2001

Contents

Revision History

Rev.	Draft/Changes	Date
—001	• Initial Release	March 2001

1. Introduction

The Floating-Point IEEE Filter for the Intel® Itanium™ processor is a function, `ieee_filter()`, that is provided to help in handling unmasked floating-point exceptions for applications running on the Itanium architecture. Its name refers to the IEEE Standard for Binary Floating-Point Arithmetic, IEEE-Std 754-1985 [1], which defines some of the entities used by the IEEE filter function: rounding modes, precision, floating-point formats, floating-point operations, and floating-point exceptions.

The IEEE filter decodes the excepting instruction and re-arranges the floating-point exception information provided by the operating system in a format easier to interpret by a user program. For example, in the case of parallel floating-point instructions, the user is presented only with the component(s) that raised unmasked floating-point exception(s). The other component (if any) is handled in the background by the IEEE Filter. For every unmasked floating-point exception, the IEEE filter invokes a user defined floating-point exception handler, which passes back to the IEEE filter the desired result for the excepting instruction before execution is continued.

In the Itanium architecture, four status fields are provided in the Floating-Point Status Register (FPSR, or Application Register 40), that contain control and status bits. The Floating-Point IEEE Filter handles unmasked floating-point exceptions raised by instructions that employ the user status field, or sf0 (status field 0).

For information on Itanium processor floating-point exceptions see *The Itanium™ Architecture Software Developer's Manual* [2], *IA-64 Floating-Point Operations and the IEEE Standard for Binary Floating-Point Arithmetic* [3], or for more details, *Itanium™ Processor Floating-point Software Assistance and Floating-point Exception Handling* [4] (Chapter 1 - Introduction, Chapter 2 – Software Assistance Faults and Traps on the Itanium™ Processor, and Chapter 3 – Conditions Causing, and Responses to Floating-Point Exceptions).

Purpose of ieee_filter

Invokes a user-defined trap handler for IEEE floating-point exceptions (also for denormal exceptions in this implementation)

Prototype

```
int ieee_filter (
    unsigned int trap_type,
    unsigned long *piip,
    unsigned long *pipsr,
    unsigned long *pfpsr,
    unsigned long *pisr,
    unsigned long *ppreds,
    unsigned long *pifs,
    void *fp_state,
    int (*handler)(fpieee_record_t *));
```

Required header: `filter.h`

Return Value

The return value of `ieee_filter()` is the value returned by the user handler.

See section 2.1, *API for the Floating-Point IEEE Filter* for complete details.

Parameters

`trap_type` – indicates whether a floating-point exception fault, or a trap is being handled

`piip` – pointer to the instruction bundle containing the excepting instruction

`pipsr` – pointer to the IPSR value

`pfpsr` – pointer to the FPSR value

`pisr` - pointer to the ISR value

`ppreds` – pointer to the predicate register value

`pifs` – pointer to the IFS register value

`fp_state` – pointer to the data structure contatining pointers to the floating-point register save area

See section 2.1, *API for the Floating-Point IEEE Filter* for complete details.

Remarks

1. Because each operating system provides information about unmasked floating-point exceptions in a different format, accessible through different system calls, a function different from the IEEE filter has to be registered (using the `sigaction()` call) as a user-level floating-point exception handler. This function, called `adjust_handler()` in section 3, *Examples*, adjusts the operating system specific exception information to the format expected by the IEEE filter, and invokes it. The `ieee_filter()` function invokes in turn a user-defined trap handler for IEEE floating-point exceptions (called `user_handler()` in section 3, *Examples*), and provides it with all relevant information (the user-defined IEEE floating-point exception handler is a parameter to `ieee_filter()`).

2. Because `ieee_filter()` accesses the `_float128_t` data type defined in `filter.h`, using `ldf.fill` and `stf.spill` instructions, all the variables of this type must be 16-byte aligned (otherwise unaligned access faults might be generated).

2. Interfacing with the Floating-Point IEEE Filter

2.1. API for the Floating-Point IEEE Filter

The application should call the Floating-Point IEEE Filter according to the following prototype:

```
int ieee_filter (
unsigned int trap_type,
unsigned long *piip,
unsigned long *pipsr,
unsigned long *pfpsr,
unsigned long *pisr,
unsigned long *ppreds,
unsigned long *pifs,
void *fp_state,
int (*handler)(fpieee_record_t *));
```

where `long` stands for the 64-bit integer data type.

The list of parameters, their types (input and/or output), and their usage are the following (for parameters which are pointers, the input or output attribute refers to the object pointed at):

`unsigned int trap_type` - (input) type of exception: 0x20 for fault, 0x21 for trap, invalid otherwise

`unsigned long *piip` - (input/output) pointer to the 64-bit quantity that contains the `IIP` value (Interruption Instruction Bundle Pointer); used to read the excepting instruction; before returning to the operating system, the IEEE filter increments this pointer for floating-point faults; for all floating-point exceptions (faults or traps), the IEEE filter places in the two least significant bits of `*piip` (i.e. in `iip`) the 2-bit value of *`ri`* (restart instruction: 0, 1, or 2); this value can be used by the operating system to set the *`ri`* bits in the `IPSR` value (restart instruction – bits 42 and 41), before executing an `rfi` (return from interruption) instruction; appending these two bits to `IIP` is possible because `IIP` has the least significant four bits equal to 0 (a bundle is 16-byte aligned); the operating system has to clear the two least significant bits in `IIP` before using the value returned by the IEEE filter. Note: the TurboLinux* operating system places the *`ei`* value (excepting instruction) in the two least significant bits of *piip, but this information is read by the IEEE filter from `*pisr` (the excepting instruction: value is 1 for MFI or MFB templates, and 2 for the MMF template)

`unsigned long *pipsr` - (output) pointer to the 64-bit quantity that contains the `IPSR` value (Interruption Processor Status Register); this is used to set in the User Mask portion (UM) the *`mfh`* and/or *`mfl`* bits (bits 5 and 4) that indicate whether a ‘‘low’’ floating-point register was modified (f2 to f31), or respectively a ‘‘high’’ one (f32 to f127); the filter also sets the value of the *`ri`* bits (bits 42 and 41 - restart instruction) in `*pipsr`. Note 1: the operating system can clear the *`mfl`* and *`mfh`* bits in the 64-bit value pointed at by `pipsr` before calling the user handler (and the IEEE filter), and can examine them upon return. If set, they indicate that some floating-point registers have been written by the user defined floating-point exception handler. Note 2: an operating system might choose to pass only the `UM` (user mask) value to the user exception handler, instead of the `IPSR` value, and thus not to use the restart

instruction information from `*pipsr`, but rather from `*piip` (this is the case for TurboLinux); in either case, an 'adjust handler' that is part of the user handler will have to create a copy of the `IPSR` (or `UM`) to pass to the IEEE filter, then retrieve the *`mfl`* and *`mfh`* bits upon return from the filter, and to perform a logical OR with the old bits from the User Mask.

`unsigned long *pfpsr` - (input/output) pointer to the 64-bit quantity that contains the `FPSR` value (Floating-Point Status Register); used to read control bit settings, and to write new status flag values; upon return, `pfpsr` points to the updated floating-point status register value (`FPSR`); only the exception masks of the user status field may be changed upon return from the call to `ieee_filter ()`

`unsigned long *pisr` - (input) pointer to the 64-bit quantity that contains the `ISR` value (Interruption Status Register); it is read to determine the type of an incoming floating-point exception, and the syllable (slot) number of the excepting instruction in the bundle, *`ei`* (0 - invalid, 1 – for MFI or MFB templates, or 2 – for MMF template); Note: the operating system places the *`ei`* value also in the two least significant bits of `iip`

`unsigned long *ppreds` - (input/output) pointer to the 64-bit quantity that contains the predicate register values; used to read the qualifying predicate of the excepting instruction; if the instruction that caused the floating-point fault or trap has at least one output that is a predicate register, then after the call to `ieee_filter()`, the value of `ppreds` points to the updated value of the 64-bit predicate register (otherwise, the predicates are unchanged)

`unsigned long *pifs` - (input) pointer to the 64-bit quantity that contains the `IFS` value (Interruption Function State); used to read the value of `CFM` (the Current Frame Marker), and to extract the values of the rotating register bases for floating-point and predicate registers

`void *fp_state` - pointer to floating-point state area of type `fp_state_t`, containing the saved values of the floating-point registers (the current floating-point register state); the definition of `fp_state_t` is as follows:

```
typedef struct  fp_state_s {
      _float128_t *fp_low; // pointer to f2-f15
      _float128_t *fp_high; // pointer to f16-f127
} fp_state_t;
```

where `_float128_t` is:

```
typedef struct {
      unsigned int W[4];
} _float128_t;
```

`int (*handler)(fpieee_record_t *)` – user defined floating-point exception handler; it has to decode the fields of the IEEE record passed to it as a parameter by the IEEE filter, and to set the desired value for the result of the excepting instruction; if execution of the user application has to be continued, the user handler has to return to the IEEE filter the constant `CONTINUE_EXECUTION`, defined in `filter.h`; the value `CONTINUE_SEARCH` means that another exception handler should be searched for; any other return value is an error indication.

The `fpieee_record_t` structure, defined in `filter.h`, contains information pertaining to an IEEE floating-point exception. This structure is passed to the user-defined trap handler by `ieee_filter()`.

Note that two functions are expected to be provided in the user space for reading and writing floating-point register values:

```
_float128_t
get_fp_register (int fp_reg, fp_state_t *fp_state);
```

and

```
void
set_fp_register (int fp_reg, _float128_t value, fp_state_t *fp_state);
```

The return code of the IEEE floating-point filter can be:

`CONTINUE_EXECUTION` - the floating-point exception was successfully handled, and a result is being provided; this value is returned only if it is also the return value from the user floating-point exception handler to the IEEE filter

`CONTINUE_SEARCH` - the floating-point exception handling was unsuccessful; this assumes that another exception handler should be searched for; if the operating system does not provide this option, then such a return value is merely an indication of an error that can be detected by the user code calling the IEEE filter

any other value – an indication of some error that occurred while trying to handle the floating-point exception; ; this value is returned only if it is also the return value from the user floating-point exception handler to the IEEE filter

2.2. Data Structures

The data types and constants presented next are defined in an include file, `filter.h`, that is required for every application that invokes the IEEE floating-point filter function.

Information is exchanged between the IEEE filter and the user floating-point exception handler through a data structure of type `fpieee_record_t` (IEEE record), defined as shown below:

```
typedef struct {
    unsigned int rounding_mode : 2;
    unsigned int precision : 3;
    unsigned int operation :12;
    fpieee_exception_flags_t cause;
    fpieee_exception_flags_t enable;
    fpieee_exception_flags_t status;
    fpieee_value_t operand1;
    fpieee_value_t operand2;
    fpieee_value_t operand3;
    fpieee_value_t result;
} fpieee_record_t;
```

All the fields except the last one (`result`) are filled in only by the IEEE filter function, which thus communicates the necessary exception information to the user handler. The last field (`result`) is used to pass information both ways – from the IEEE filter to the user handler, and back. The IEEE filter function ensures that this value provided by the user handler will be used as the result of the excepting instruction when execution of the user application is continued. The types and formats of the various fields in the IEEE record are specified below. Programming examples are also included in section 3, *Examples*.

The type of the `rounding_mode` field is defined by the following '`enum`'. The four entries correspond to the four IEEE rounding modes for floating-point operations: rounding to nearest, to negative infinity, to positive infinity, and toward zero:

```
typedef enum {
    _fp_round_nearest,
    _fp_round_minus_infinity,
    _fp_round_plus_infinity,
    _fp_round_chopped
```

```
} fpieee_rounding_mode_t;
```

The `precision` field has the following type, which contains entries for the three most common IEEE floating-point data types: single precision (with 24-bit significands), double precision (with 53-bit significands), and double-extended precision (with 64-bit significands):

```
typedef enum {
    _fp_precision24,
    _fp_precision53,
    _fp_precision64,
} _fpieee_precision_t;
```

The `operation` field defines the possible floating-point operations that can raise unmasked exceptions. Besides add, subtract, multiply, divide, square root, compare, conversion from floating-point to integer, and conversion with truncation, this list includes also entries for three-operand instructions from the floating-point multiply-add family defined in the Itanium architecture. These operations are `fma` (calculates a * b + c with only rounding error), `fms` (calculates a * b – c), and `fnma` (calculates –a * b + c). Separate entries are present for `fma.s`, `fms.s`, `fnma.s` (single-precision operation specified statically), `fma.d`, `fms.d`, `fnma.d` (double precision operation specified statically), and `fma`, `fms`, `fnma` (the precision is specified dynamically by fields of the Floating-Point Status Register). The other Itanium processor instructions that have entries defined in the enumeration below are `fmax` (floating-point maximum), `fmin` (floating-point minimum), `famax` (floating-point absolute maximum), and `famin` (floating-point absolute minimum).

```
typedef enum {
    _fp_code_unspecified,
    _fp_code_add,
    _fp_code_subtract,
    _fp_code_multiply,
    _fp_code_divide,
    _fp_code_square_root,
    _fp_code_compare,
    _fp_code_convert,
    _fp_code_convert_trunc
    _fp_code_fma,
    _fp_code_fma_single,
    _fp_code_fma_double,
    _fp_code_fms,
    _fp_code_fms_single,
    _fp_code_fms_double,
    _fp_code_fnma,
    _fp_code_fnma_single,
    _fp_code_fnma_double,
    _fp_code_fmin,
    _fp_code_fmax,
    _fp_code_famin,
    _fp_code_famax
} fp_operation_code_t;
```

The next three fields in the IEEE record, `cause`, `enable`, and `status,` are all of type `fpieee_exception_flags_t,` defined below. The `cause` field specifies the cause of the floating-point exception. This will correspond to an IEEE-defined floating-point fault (invalid operation or divide-by-zero), or to a floating-point trap (overflow, underflow, or inexact result. Note that the denormal exceptions (floating-point faults) that can be raised by Itanium processor (as well as IA-32) floating-point instructions are not defined by the IEEE standard. Therefore, it is implicitly assumed that a denormal exception was raised whenever the user-defined exception handler is invoked by the IEEE filter with no cause bit set. The `enable` field specifies which of the five IEEE floating-point exceptions are

enabled (or unmasked). The `status` field specifies whether the five IEEE floating-point status flags are set or not. Note that the IEEE record does not specify whether the denormal exceptions are enabled or not, nor the value of the denormal status flag.

```
typedef struct {
    unsigned int inexact_result : 1;
    unsigned int underflow : 1;
    unsigned int overflow : 1;
    unsigned int zero_divide : 1;
    unsigned int invalid_operation : 1;
} fpieee_exception_flags_t;
```

The last four fields in the IEEE record correspond to the excepting floating-point instruction operands (up to three – `operand1`, `operand2`, and `operand3`), and to its result (the `result` field). The operand information is passed by the IEEE filter to the user floating-point exception handler. For the result, the `operand_valid` and `format` fields are always passed by the IEEE filter to the user handler. The information in the `value` field depends on the type of floating-point exception. For floating-point faults, no information is passed in this field from the IEEE filter to the user handler, which will fill in the desired value before returning to the IEEE filter. For floating-point traps, a result value is provided to the user handler by the IEEE filter (as specified by the IEEE Standard for Binary Floating-Point Arithmetic), but the user handler can also set a new value in this field before returning to the IEEE filter. All four entries for operands and result have the type `fpieee_value_t` defined as shown below.

```
typedef struct {
    union {
        float                   fp32_value;
        double                  fp64_value;
        float80_t               fp80_value;
        _float128_t             fp82_value;
        int                     i32_value;
        long                    i64_value;
        unsigned int            u32_value;
        unsigned long           u64_value;
        fpieee_compare_result_t compare_value;
    } value;
    unsigned int operand_valid : 1;
    fpieee_format_t format : 4;
} fpieee_value_t;
```

Some of the value field types are also defined in `filter.h`. These are `float80_t` for double-extended floating-point values, `_float128_t` for 82-bit register file format floating-point values, , and `fpieee_compare_result_t` for the result value of a comparison operation:

```
typedef struct {
    unsigned short W[5];
} float80_t;
typedef struct {
    unsigned int W[4];
} _float128_t;


typedef enum {
    _fp_compare_equal,
    _fp_compare_greater,
```

```
    _fp_compare_less,
    _fp_compare_unordered
} fpieee_compare_result_t;
```

The `compare_value` field can be used to specify one of four possible values for the result of a compare operation: equal, less than, greater than, or unordered.

Finally, the `format` field has type `fpieee_format_t`:

```
typedef enum {
    _fp_format_fp32, /*1-bit sign+8-bit exp.+24-bit signif. */
    _fp_format_fp64, /*1-bit sign+11-bit exp.+53-bit signif.*/
    _fp_format_fp80, /*1-bit sign+15-bit exp.+64-bit signif.*/
    _fp_format_fp82, /*1-bit sign+17-bit exp.+64-bit signif.*/
    _fp_format_i32,  /* 32-bit integer                      */
    _fp_format_i64,  /* 64-bit integer                      */
    _fp_format_u32,  /* 32-bit unsigned integer             */
    _fp_format_u64,  /* 64-bit unsigned integer             */
    _fp_format_compare, /* compare value format             */
} fpieee_format_t;
```

3. Examples

The following three examples illustrate raising and handling unmasked floating-point exceptions in TurboLinux, running on the Itanium processor.

The first example illustrates floating-point exceptions raised by scalar (non-parallel) floating-point instrructions from assembly language routines, called from C code. The second example is similar, but contains only C code. The third example illustrates exceptions raised by parallel floating-point instrructions from assembly language routines, called from C code.

Note that the preferred method of reading or writing floating-point status and control information is by using functions provided by the operating system. The following examples achieve the same result by reading and writing directly the FPSR.

3.1. Example 1. Floating-Point Exceptions Raised by Scalar Instructions from Assembly Language Code

The first example illustrates how to use the IEEE filter to handle unmasked floating-point exceptions raised by Itanium processor instructions. The information supplied by the operating system needs to be adjusted to the format expected by the IEEE filter. This is achieved through a function named `adjust_handler()` – the first one presented in the example below. The `adjust_handler()`

function is the floating-point exception handler that has to be registered by the application via the `sigaction()` system call, as seen in the `main()` function further . The actual user handler is `user_handler()`, which covers the most common cases of floating-point exceptions that can occur in Itanium processor applications. Function `main()` registers the floating-point exception handler and initializes the operand values and the FPSR value for the last function presented below, `run_fma()`, used to trigger the five possible exceptions for the `fma.s` instruction: invalid operation, denormal operand, overflow, underflow, and inexact operation. Triggering floating-point exceptions could be done directly from C, as seen further in Example 2.

When a floating-point exception occurs, the registered handler – adjust_handler() - is invoked. This function calls in turn ieee_filter(), passing to it as one of its parameters, a pointer to the user defined handler – user_handler(). This last function does the actual work of reading the IEEE record and supplying a result for the excepting instruction.

```
adjust_handler (int sig, siginfo_t *sinfo, struct sigcontext *pcontext) {

  fp_state_t fp_state;
  int retval;
  unsigned int type;
  unsigned long um_copy;

  if (sig != SIGFPE) {
    printf ("ERROR in adjust_handler (): sig != SIGFPE, sig = %x\n", sig);
    exit (1);
  }


  /* set up the floating-point area */
  fp_state.fp_low = (_float128_t *)&(pcontext->sc_fr[2]);
  fp_state.fp_high = (_float128_t *)&(pcontext->sc_fr[32]);

  type = ((sinfo->_sifields._sigfault._isr & 0x0000ff80) == 0) ? 0x20 : 0x21;
      /* 0x20 fault, 0x21 trap */
  um_copy = pcontext->sc_um & 0x30;

  retval = ieee_filter (
    type,
    &pcontext->sc_ip,
    &um_copy,
    &pcontext->sc_ar_fpsr,
    &sinfo->_sifields._sigfault._isr,
    &pcontext->sc_pr,
    &pcontext->sc_cfm,
    &fp_state,
    user_handler); /* user_handler() is user defined */

  pcontext->sc_um |= (um_copy & 0x30);
  if (retval == CONTINUE_EXECUTION) {
    ; /* it was an IEEE floating-point exception - let execution to continue */
  } else {
    fprintf (stderr,"ERROR: Exception not handled by user exception handler\n");
    exit (1);
  }
  /* Note that the floating-point area does not require restoring - it was
   * accessed by pointer */
  return;
```

Two functions are used to get/set floating-point values from/into the floating-point register save area in memory:

```
/* function used by the filter to read floating-point register values */
_float128_t
```

```
get_fp_register (int fp_reg, fp_state_t *fp_state) {
  _float128_t retval;
  if (fp_reg == 0) {
    retval.W[3] = (unsigned int)0;
    retval.W[2] = (unsigned int)0;
    retval.W[1] = (unsigned int)0;
    retval.W[0] = (unsigned int)0;
  } else if (fp_reg == 1) {
    retval.W[3] = (unsigned int)0;
    retval.W[2] = (unsigned int)0xffff;
    retval.W[1] = (unsigned int)0x80000000;
    retval.W[0] = (unsigned int)0x00000000;
  } else if (fp_reg < 32) {
    retval = fp_state->fp_low[fp_reg - 2];
  } else {
    retval = fp_state->fp_high[fp_reg - 32];
  }
  return (retval);
}

/* function used by the filter to write floating-point register values */
void
set_fp_register (int fp_reg, _float128_t value, fp_state_t *fp_state) {
  if (fp_reg == 0 || fp_reg == 1) {
    ; /* do nothing */
  } else if (fp_reg < 32) {
    fp_state->fp_low[fp_reg - 2] = value;
  } else {
    fp_state->fp_high[fp_reg - 32] = value;
  }
}
```

The `main()` function registers the floating-point exception handler and initializes the operand values and the FPSR value for the last function presented below, `run_fma()`. The user handler, `user_handler()`, interprets the exception information passed to it by the IEEE filter, and sets the result for the excepting instruction. It can also mask (disable) or unmask (enable) IEEE floating-point exceptions.

```
void
main ()

{

  unsigned long fpsr;
  struct sigaction act, oact;
  unsigned int opd1, opd2, opd3, res;

  // Example for raising and handling unmasked floating-point exceptions;
  // an assembly coded function, run_fma (), executes fma.s.s0 f6 = f7, f8, f9,
  // with pointers to the FPSR and to the single precision parameters passed by
  // the caller (the single precision floating-point values are specified by
  // their hexadecimal representation, as unsigned integers); passing the FPSR
  // value is necessary in order to selectively unmask (enable) floating-point
  // exceptions

  /* register user floating-point exception handler */
  act.sa_handler = (void (*)(int))adjust_handler;
  act.sa_flags = SA_NOMASK;
  sigaction (SIGFPE, &act, &oact);

  // IMPORTANT NOTE: when setting the FPSR, always leave status field 1 (sf1)
  // at its default value (traps disabled, rounding-to-nearest, 64-bit
  // precision, widest range exponent set, flush-to-zero mode not set),
  // otherwise floating-point divide, square root, remainder, and libm
  // transcendental function calculations, as well as integer divide and
  // remainder operations might generate incorrect results

  /* invalid exception */
  printf ("\n******************* INVALID EXCEPTION ******************* \n");
  fpsr = 0x0009804c0270033e; /* default FPSR, with I exceptions unmasked */
  opd1 = 0x00000000; /* 0.0 */
```

```
  opd2 = 0x7f800000; /* +infinity */
  opd3 = 0x00000000; /* 0.0 */
  res = 0x00000000; /* 0.0 */
  run_fma (&fpsr, &res, &opd1, &opd2, &opd3);
  printf ("after invalid exception res = %8.8x\n", res);

  /* denormal exception */
  printf ("\n******************* DENORMAL EXCEPTION ****************** \n");
  fpsr = 0x0009804c0270033d; /* default FPSR, with D exceptions unmasked */
  opd1 = 0x3f800000; /* 1.0 */
  opd2 = 0x00000001; /* smallest positive denormal */
  opd3 = 0x00000000; /* 0.0 */
  res = 0x00000000; /* 0.0 */
  run_fma (&fpsr, &res, &opd1, &opd2, &opd3);
  printf ("after denormal exception res = %8.8x\n", res);

  /* overflow exception */
  printf ("\n******************* OVERFLOW EXCEPTION ****************** \n");
  fpsr = 0x0009804c02700337; /* default FPSR, with O exceptions unmasked */
  opd1 = 0x7f7fffff; /* 1.1...1 * 2^127 */
  opd2 = 0x7f7fffff; /* 1.1...1 * 2^127 */
  opd3 = 0x00000000; /* 0.0 */
  res = 0x00000000; /* 0.0 */
  run_fma (&fpsr, &res, &opd1, &opd2, &opd3);
  printf ("after overflow exception res = %8.8x\n", res);

  /* underflow exception */
  printf ("\n******************* UNDERFLOW EXCEPTION ****************** \n");
  fpsr = 0x0009804c0270032f; /* default FPSR, with U exceptions unmasked */
  opd1 = 0x00800000; /* smallest positive normal */
  opd2 = 0x00800000; /* smallest positive normal */
  opd3 = 0x00000000; /* 0.0 */
  res = 0x00000000; /* 0.0 */
  run_fma (&fpsr, &res, &opd1, &opd2, &opd3);
  printf ("after underflow exception res = %8.8x\n", res);

  /* inexact exception */
  printf ("\n******************* INEXACT EXCEPTION ****************** \n");
  fpsr = 0x0009804c0270031f; /* default FPSR, with P exceptions unmasked */
  opd1 = 0x3f800001; /* 1.0 + 1 ulp */
  opd2 = 0x3f800001; /* 1.0 + 1 ulp */
  opd3 = 0x00000000; /* 0.0 */
  res = 0x00000000; /* 0.0 */
  run_fma (&fpsr, &res, &opd1, &opd2, &opd3);
  printf ("after inexact exception res = %8.8x\n", res);

}

int
user_handler (fpieee_record_t *ieee) {

  float f32;
  double f64;
  float80_t f80;
  _float128_t f82;
  int i32;
  long i64;
  unsigned int u32;
  unsigned long u64;

  printf ("\n *** BEGIN USER HANDLER ***\n\n");

  if (ieee->rounding_mode == _fp_round_nearest)
    printf ("USER HANDLER: round to nearest\n");
  else if (ieee->rounding_mode == _fp_round_minus_infinity)
    printf ("USER HANDLER: round to negative infinity\n");
  else if (ieee->rounding_mode == _fp_round_plus_infinity)
    printf ("USER HANDLER: round to positive infinity\n");
  else if (ieee->rounding_mode == _fp_round_chopped)
    printf ("USER HANDLER: round to zero\n");
  else
    printf ("USER HANDLER: unknown rounding mode\n");
```

```
if (ieee->precision == _fp_precision24)
  printf ("USER HANDLER: single precision\n");
else if (ieee->precision == _fp_precision53)
  printf ("USER HANDLER: double precision\n");
else if (ieee->precision == _fp_precision64)
  printf ("USER HANDLER: double extended precision\n");
else
  printf ("USER HANDLER: unknown precision mode\n");

if (ieee->operation == _fp_code_add)
  printf ("USER HANDLER: add operation\n");
else if (ieee->operation == _fp_code_subtract)
  printf ("USER HANDLER: subtract operation\n");
 else if (ieee->operation == _fp_code_multiply)
  printf ("USER HANDLER: multiply operation\n");
else if (ieee->operation == _fp_code_divide)
  printf ("USER HANDLER: divide operation\n");
else if (ieee->operation == _fp_code_square_root)
  printf ("USER HANDLER: square root operation\n");
else if (ieee->operation == _fp_code_compare)
  printf ("USER HANDLER: compare operation\n");
else if (ieee->operation == _fp_code_convert)
  printf ("USER HANDLER: convert operation\n");
else if (ieee->operation == _fp_code_convert_trunc)
  printf ("USER HANDLER: convert and truncate operation\n");
else if (ieee->operation == _fp_code_fma)
  printf ("USER HANDLER: floating-point multiply-add  operation\n");
else if (ieee->operation == _fp_code_fma_single)
  printf ("USER HANDLER: single floating-point multiply-add operation\n");
else if (ieee->operation == _fp_code_fma_double)
  printf ("USER HANDLER: double floating-point multiply-add  operation\n");
else if (ieee->operation == _fp_code_fms)
  printf ("USER HANDLER: floating-point multiply-subtract  operation\n");
else if (ieee->operation == _fp_code_fms_single)
  printf ("USER HANDLER: single floating-point multiply-subtract "
     "operation\n");
else if (ieee->operation == _fp_code_fms_double)
  printf ("USER HANDLER: double floating-point multiply-subtract "
     "operation\n");
else if (ieee->operation == _fp_code_fnma)
  printf ("USER HANDLER: negative floating-point multiply-add "
     "operation\n");
else if (ieee->operation == _fp_code_fnma_single)
  printf ("USER HANDLER: single negative floating-point multiply-add "
     "operation\n");
else if (ieee->operation == _fp_code_fnma_double)
  printf ("USER HANDLER: negative double floating-point multiply-add "
     "operation\n");
else if (ieee->operation == _fp_code_fmin)
  printf ("USER HANDLER: floating-point minimum operation\n");
else if (ieee->operation == _fp_code_fmax)
  printf ("USER HANDLER: floating-point maximum operation\n");
else if (ieee->operation == _fp_code_famin)
  printf ("USER HANDLER: floating-point absolute minimum operation\n");
else if (ieee->operation == _fp_code_famax)
  printf ("USER HANDLER: floating-point absolute maximum operation\n");
else if (ieee->operation == _fp_code_unspecified)
  printf ("USER HANDLER: unspecified floating-point operation\n");
else
  printf ("USER HANDLER: unknown floating-point operation\n");

printf ("USER HANDLER: cause bits PUOZI = %1.1x %1.1x %1.1x %1.1x %1.1x\n",
    ieee->cause.inexact_result, ieee->cause.underflow, ieee->cause.overflow,
    ieee->cause.zero_divide, ieee->cause.invalid_operation);

printf ("USER HANDLER: enable bits PUOZI = %1.1x %1.1x %1.1x %1.1x %1.1x\n",
    ieee->enable.inexact_result, ieee->enable.underflow,
    ieee->enable.overflow, ieee->enable.zero_divide,
    ieee->enable.invalid_operation);

printf ("USER HANDLER: status bits PUOZI = %1.1x %1.1x %1.1x %1.1x %1.1x\n",
    ieee->status.inexact_result, ieee->status.underflow,
    ieee->status.overflow, ieee->status.zero_divide,
```

```
    ieee->status.invalid_operation);

if (ieee->operand1.operand_valid) {
  printf ("USER HANDLER: operand1 is valid\n");
  if (ieee->operand1.format == _fp_format_fp32) {
    printf ("USER HANDLER: operand1 has format _fp_format_fp32\n");
    f32 = ieee->operand1.value.fp32_value;
    printf ("USER HANDLER: operand1 value = %8.8x\n", *(unsigned int *)&f32);
  } else if (ieee->operand1.format == _fp_format_fp64) {
    printf ("USER HANDLER: operand1 has format _fp_format_fp64\n");
    f64 = ieee->operand1.value.fp64_value;
    printf ("USER HANDLER: operand1 value = %Lx\n",
        *(unsigned long long *)&f64);
  } else if (ieee->operand1.format == _fp_format_fp80) {
    printf ("USER HANDLER: operand1 has format _fp_format_fp80\n");
    f80 = ieee->operand1.value.fp80_value;
    printf ("USER HANDLER: operand1 value = %4.4x%4.4x%4.4x%4.4x%4.4x\n",
        f80.W[4], f80.W[3], f80.W[2], f80.W[1], f80.W[0]);
  } else if (ieee->operand1.format == _fp_format_fp82) {
    printf ("USER HANDLER: operand1 has format _fp_format_fp82\n");
    f82 = ieee->operand1.value.fp82_value;
    printf ("USER HANDLER: operand1 value = %8.8x%8.8x%8.8x%8.8x\n",
        f82.W[3], f82.W[2], f82.W[1], f82.W[0]);
  } else if (ieee->operand1.format == _fp_format_i32) {
    printf ("USER HANDLER: operand1 has format _fp_format_i32\n");
    i32 = ieee->operand1.value.i32_value;
    printf ("USER HANDLER: operand1 value = %8.8x\n", i32);
  } else if (ieee->operand1.format == _fp_format_i64) {
    printf ("USER HANDLER: operand1 has format _fp_format_i64\n");
    i64 = ieee->operand1.value.i64_value;
    printf ("USER HANDLER: operand1 value = %Lx\n",
        (unsigned long long)i64);
  } else if (ieee->operand1.format == _fp_format_u32) {
    printf ("USER HANDLER: operand1 has format _fp_format_u32\n");
    u32 = ieee->operand1.value.u32_value;
    printf ("USER HANDLER: operand1 value = %8.8x\n", u32);
  } else if (ieee->operand1.format == _fp_format_u64) {
    printf ("USER HANDLER: operand1 has format _fp_format_u64\n");
    u64 = ieee->operand1.value.u64_value;
    printf ("USER HANDLER: operand1 value = %Lx\n", (unsigned long long)u64);
  } else if (ieee->operand1.format == _fp_format_compare) {
    printf ("USER HANDLER: operand1 has format _fp_format_compare "
        "(invalid)\n");
  } else {
    printf ("USER HANDLER: operand1 has unknown format\n");
  }
} else {
  printf ("USER HANDLER: operand1 is not valid\n");
}

if (ieee->operand2.operand_valid) {
  printf ("USER HANDLER: operand2 is valid\n");
  if (ieee->operand2.format == _fp_format_fp32) {
    printf ("USER HANDLER: operand2 has format _fp_format_fp32\n");
    f32 = ieee->operand2.value.fp32_value;
    printf ("USER HANDLER: operand2 value = %8.8x\n", *(unsigned int *)&f32);
  } else if (ieee->operand2.format == _fp_format_fp64) {
    printf ("USER HANDLER: operand2 has format _fp_format_fp64\n");
    f64 = ieee->operand2.value.fp64_value;
    printf ("USER HANDLER: operand2 value = %Lx\n",
        *(unsigned long long *)&f64);
  } else if (ieee->operand2.format == _fp_format_fp80) {
    printf ("USER HANDLER: operand2 has format _fp_format_fp80\n");
    f80 = ieee->operand2.value.fp80_value;
    printf ("USER HANDLER: operand2 value = %4.4x%4.4x%4.4x%4.4x%4.4x\n",
        f80.W[4], f80.W[3], f80.W[2], f80.W[1], f80.W[0]);
  } else if (ieee->operand2.format == _fp_format_fp82) {
    printf ("USER HANDLER: operand2 has format _fp_format_fp82\n");
    f82 = ieee->operand2.value.fp82_value;
    printf ("USER HANDLER: operand2 value = %8.8x%8.8x%8.8x%8.8x\n",
        f82.W[3], f82.W[2], f82.W[1], f82.W[0]);
  } else if (ieee->operand2.format == _fp_format_i32) {
    printf ("USER HANDLER: operand2 has format _fp_format_i32\n");
```

```
    i32 = ieee->operand2.value.i32_value;
    printf ("USER HANDLER: operand2 value = %8.8x\n", i32);
  } else if (ieee->operand2.format == _fp_format_i64) {
    printf ("USER HANDLER: operand2 has format _fp_format_i64\n");
    i64 = ieee->operand2.value.i64_value;
    printf ("USER HANDLER: operand2 value = %Lx\n", (unsigned long long)i64);
  } else if (ieee->operand2.format == _fp_format_u32) {
    printf ("USER HANDLER: operand2 has format _fp_format_u32\n");
    u32 = ieee->operand2.value.u32_value;
    printf ("USER HANDLER: operand2 value = %8.8x\n", u32);
  } else if (ieee->operand2.format == _fp_format_u64) {
    printf ("USER HANDLER: operand2 has format _fp_format_u64\n");
    u64 = ieee->operand2.value.u64_value;
    printf ("USER HANDLER: operand2 value = %Lx\n", (unsigned long long)u64);
  } else if (ieee->operand2.format == _fp_format_compare) {
    printf ("USER HANDLER: operand2 has format _fp_format_compare "
        "(invalid)\n");
  } else {
    printf ("USER HANDLER: operand2 has unknown format\n");
  }
} else {
  printf ("USER HANDLER: operand2 is not valid\n");
}

if (ieee->operand3.operand_valid) {
  printf ("USER HANDLER: operand3 is valid\n");
  if (ieee->operand3.format == _fp_format_fp32) {
    printf ("USER HANDLER: operand3 has format _fp_format_fp32\n");
    f32 = ieee->operand3.value.fp32_value;
    printf ("USER HANDLER: operand3 value = %8.8x\n", *(unsigned int *)&f32);
  } else if (ieee->operand3.format == _fp_format_fp64) {
    printf ("USER HANDLER: operand3 has format _fp_format_fp64\n");
    f64 = ieee->operand3.value.fp64_value;
    printf ("USER HANDLER: operand3 value = %Lx\n",
        *(unsigned long long *)&f64);
  } else if (ieee->operand3.format == _fp_format_fp80) {
    printf ("USER HANDLER: operand3 has format _fp_format_fp80\n");
    f80 = ieee->operand3.value.fp80_value;
    printf ("USER HANDLER: operand3 value = %4.4x%4.4x%4.4x%4.4x%4.4x\n",
        f80.W[4], f80.W[3], f80.W[2], f80.W[1], f80.W[0]);
  } else if (ieee->operand3.format == _fp_format_fp82) {
    printf ("USER HANDLER: operand3 has format _fp_format_fp82\n");
    f82 = ieee->operand3.value.fp82_value;
    printf ("USER HANDLER: operand3 value = %8.8x%8.8x%8.8x%8.8x\n",
        f82.W[3], f82.W[2], f82.W[1], f82.W[0]);
  } else if (ieee->operand3.format == _fp_format_i32) {
    printf ("USER HANDLER: operand3 has format _fp_format_i32\n");
    i32 = ieee->operand3.value.i32_value;
    printf ("USER HANDLER: operand3 value = %8.8x\n", i32);
  } else if (ieee->operand3.format == _fp_format_i64) {
    printf ("USER HANDLER: operand3 has format _fp_format_i64\n");
    i64 = ieee->operand3.value.i64_value;
    printf ("USER HANDLER: operand3 value = %Lx\n", (unsigned long long)i64);
  } else if (ieee->operand3.format == _fp_format_u32) {
    printf ("USER HANDLER: operand3 has format _fp_format_u32\n");
    u32 = ieee->operand3.value.u32_value;
    printf ("USER HANDLER: operand3 value = %8.8x\n", u32);
  } else if (ieee->operand3.format == _fp_format_u64) {
    printf ("USER HANDLER: operand3 has format _fp_format_u64\n");
    u64 = ieee->operand3.value.u64_value;
    printf ("USER HANDLER: operand3 value = %Lx\n", (unsigned long long)u64);
  } else if (ieee->operand3.format == _fp_format_compare) {
    printf ("USER HANDLER: operand3 has format _fp_format_compare "
        "(invalid)\n");
  } else {
    printf ("USER HANDLER: operand3 has unknown format\n");
  }
} else {
  printf ("USER HANDLER: operand3 is not valid\n");
}

if (ieee->result.operand_valid) {
  printf ("USER HANDLER: result is valid\n");
```

```
    if (ieee->result.format == _fp_format_fp32) {
      printf ("USER HANDLER: result has format _fp_format_fp32\n");
      f32 = ieee->result.value.fp32_value;
      printf ("USER HANDLER: result value = %8.8x\n", *(unsigned int *)&f32);
    } else if (ieee->result.format == _fp_format_fp64) {
      printf ("USER HANDLER: result has format _fp_format_fp64\n");
      f64 = ieee->result.value.fp64_value;
      printf ("USER HANDLER: result value = %Lx\n", *(unsigned long long*)&f64);
    } else if (ieee->result.format == _fp_format_fp80) {
      printf ("USER HANDLER: result has format _fp_format_fp80\n");
      f80 = ieee->result.value.fp80_value;
      printf ("USER HANDLER: result value = %4.4x%4.4x%4.4x%4.4x%4.4x\n",
          f80.W[4], f80.W[3], f80.W[2], f80.W[1], f80.W[0]);
    } else if (ieee->result.format == _fp_format_fp82) {
      printf ("USER HANDLER: result has format _fp_format_fp82\n");
      f82 = ieee->result.value.fp82_value;
      printf ("USER HANDLER: result value = %8.8x%8.8x%8.8x%8.8x\n",
          f82.W[3], f82.W[2], f82.W[1], f82.W[0]);
    } else if (ieee->result.format == _fp_format_i32) {
      printf ("USER HANDLER: result has format _fp_format_i32\n");
      i32 = ieee->result.value.i32_value;
      printf ("USER HANDLER: result value = %8.8x\n", i32);
    } else if (ieee->result.format == _fp_format_i64) {
      printf ("USER HANDLER: result has format _fp_format_i64\n");
      i64 = ieee->result.value.i64_value;
      printf ("USER HANDLER: result value = %Lx\n", (unsigned long long)i64);
    } else if (ieee->result.format == _fp_format_u32) {
      printf ("USER HANDLER: result has format _fp_format_u32\n");
      u32 = ieee->result.value.u32_value;
      printf ("USER HANDLER: result value = %8.8x\n", u32);
    } else if (ieee->result.format == _fp_format_u64) {
      printf ("USER HANDLER: result has format _fp_format_u64\n");
      u64 = ieee->result.value.u64_value;
      printf ("USER HANDLER: result value = %Lx\n", (unsigned long long)u64);
    } else if (ieee->result.format == _fp_format_compare) {
      printf ("USER HANDLER: result has format _fp_format_compare\n");
    } else {
      printf ("USER HANDLER: result has unknown format\n");
    }
  } else {
    printf ("USER HANDLER: result is not valid\n");
  }

  // the result should be set according to the excepting operation, and the
  // result format; in this example, it will be assumed that the result is a
  // single precision floating-point number
  if (ieee->cause.invalid_operation) {
    ieee->result.value.u32_value = 0x3f800001; /* 1.0 + 1 ulp */
    printf ("USER HANDLER: invalid exception\n");
  } else if (ieee->cause.zero_divide) {
    ieee->result.value.u32_value = 0x3f800002; /* 1.0 + 2 ulp */
    printf ("USER HANDLER: zero divide exception\n");
  } else if (ieee->cause.overflow) {
    ieee->result.value.u32_value = 0x3f800003; /* 1.0 + 3 ulp */
    printf ("USER HANDLER: overflow exception\n");
  } else if (ieee->cause.underflow) {
    ieee->result.value.u32_value = 0x3f800004; /* 1.0 + 4 ulp */
    printf ("USER HANDLER: underflow exception\n");
  } else if (ieee->cause.inexact_result) {
    ieee->result.value.u32_value = 0x3f800005; /* 1.0 + 5 ulp */
    printf ("USER HANDLER: inexact exception\n");
  } else { /* cause_denormal is assumed */
    ieee->result.value.u32_value = 0x3f800006; /* 1.0 + 6 ulp */
    printf ("USER HANDLER: denormal exception\n");
  }

  printf ("\n *** END USER HANDLER ***\n\n");

  return CONTINUE_EXECUTION;
}
```

The assembly language function used to trigger the floating-point exceptions is `run_fma()`:

```
run_fma:
  alloc r31 = ar.pfs,5,2,0,0  // r32, r33, r34, r35, r36, r37, r38

  // &fpsr is in r32
  // &res (output) is in r33
  // &opd1 (input) is in r34
  // &opd2 (input) is in r35
  // &opd3 (input) is in r36

  mov r38 = ar40;; // save old FPSR in r38
  ld8 r37 = [r32];; // load new FPSR in r37
  mov ar40 = r37;; // set new value of FPSR
  ldfs f7 = [r34] // load first input argument into f7
  ldfs f8 = [r35] // load second input argument into f8
  ldfs f9 = [r36];; // load third input argument into f9
  fma.s.s0 f6 = f7, f8, f9;; // f6 = f7 * f8 + f9
  mov r37 = ar40;; // store new FPSR
  st8 [r32] = r37;;
  stfs [r33] = f6 // store result
  mov ar40 = r38;; // restore original FPSR
  br.ret.sptk b0 // return
  .endp run_fma
```

The output for this first example is:

```
******************** INVALID EXCEPTION ********************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 0 0 1
USER HANDLER: enable bits PUOZI = 0 0 0 0 1
USER HANDLER: status bits PUOZI = 0 0 0 0 1
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _fp_format_fp82
USER HANDLER: operand1 value = 00000000000000000000000000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _fp_format_fp82
USER HANDLER: operand2 value = 000000000001ffff8000000000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _fp_format_fp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is not valid
USER HANDLER: invalid exception

 *** END USER HANDLER ***

after invalid exception res = 3f800001

******************** DENORMAL EXCEPTION ********************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 0 0 0
USER HANDLER: enable bits PUOZI = 0 0 0 0 0
USER HANDLER: status bits PUOZI = 0 0 0 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _fp_format_fp82
USER HANDLER: operand1 value = 0000000000000ffff8000000000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _fp_format_fp82
USER HANDLER: operand2 value = 0000000000000ff810000010000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _fp_format_fp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
```

```
USER HANDLER: result is not valid
USER HANDLER: denormal exception

 *** END USER HANDLER ***

after denormal exception res = 3f800006

******************** OVERFLOW EXCEPTION ********************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 1 0 0
USER HANDLER: enable bits PUOZI = 0 0 1 0 0
USER HANDLER: status bits PUOZI = 1 0 1 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _fp_format_fp82
USER HANDLER: operand1 value = 000000000001007effffff0000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _fp_format_fp82
USER HANDLER: operand2 value = 000000000001007effffff0000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _fp_format_fp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is valid
USER HANDLER: result has format _fp_format_fp32
USER HANDLER: result value = 5f7ffffe
USER HANDLER: overflow exception

 *** END USER HANDLER ***

after overflow exception res = 3f800003

******************** UNDERFLOW EXCEPTION ********************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 1 0 0 0
USER HANDLER: enable bits PUOZI = 0 1 0 0 0
USER HANDLER: status bits PUOZI = 0 1 0 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _fp_format_fp82
USER HANDLER: operand1 value = 0000000000000ff818000000000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _fp_format_fp82
USER HANDLER: operand2 value = 0000000000000ff818000000000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _fp_format_fp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is valid
USER HANDLER: result has format _fp_format_fp32
USER HANDLER: result value = 21800000
USER HANDLER: underflow exception

 *** END USER HANDLER ***

after underflow exception res = 3f800004

******************** INEXACT EXCEPTION ********************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 1 0 0 0 0
USER HANDLER: enable bits PUOZI = 1 0 0 0 0
USER HANDLER: status bits PUOZI = 1 0 0 0 0
```

```
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _fp_format_fp82
USER HANDLER: operand1 value = 0000000000000ffff8000010000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _fp_format_fp82
USER HANDLER: operand2 value = 0000000000000ffff8000010000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _fp_format_fp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is valid
USER HANDLER: result has format _fp_format_fp32
USER HANDLER: result value = 3f800002
USER HANDLER: inexact exception

 *** END USER HANDLER ***

after inexact exception res = 3f800005
```

3.2. Example 2. Floating-Point Exceptions Raised by Scalar Instructions from C Language Code

The second example illustrates usage of the IEEE filter for handling unmasked floating-point exceptions raised by scalar Itanium processor instructions, directly from C language code. The `adjust_handler()`, `get_fp_register()`, `set_fp_register()`, and `user_handler()` functions are the same as in Example 1. The output is also the same. The `main()` function is different, and a new function, `write_fpsr()` is used to unmask floating-point exceptions (instead of the `run_fma()` function):

```
void
main () {
  unsigned long fpsr;
  struct sigaction act, oact;
  unsigned int opd1, opd2, res;
  float fopd1, fopd2, fres;

  // Example for raising and handling unmasked floating-point exceptions from
  // a C program; the FPSR is written in order to selectively unmask (enable)
  // floating-point exceptions

  /* register user floating-point exception handler */
  act.sa_handler = (void (*)(int))adjust_handler;
  act.sa_flags = SA_NOMASK;
  sigaction (SIGFPE, &act, &oact);

  // IMPORTANT NOTE: when setting the FPSR, always leave status field 1 (sf1)
  // at its default value (traps disabled, rounding-to-nearest, 64-bit
  // precision, widest range exponent set, flush-to-zero mode not set),
  // otherwise floating-point divide, square root, remainder, and libm
  // transcendental function calculations, as well as integer divide and
  // remainder operations might generate incorrect results

  /* invalid exception */
  printf ("\n******************* INVALID EXCEPTION ****************** \n");
  fpsr = 0x0009804c0270033e; /* default FPSR, with I exceptions unmasked */
  opd1 = 0x00000000; /* 0.0 */
  fopd1 = *(float *)&opd1;
  opd2 = 0x7f800000; /* +infinity */
  fopd2 = *(float *)&opd2;
  res = 0x00000000; /* 0.0 */
  fres = *(float *)&res;
  write_fpsr (fpsr);
  fres = fopd1 * fopd2;
```

```
  res = *(unsigned int *)&fres;
    printf ("after invalid exception res = %8.8x\n", res);

    /* denormal exception */
    printf ("\n******************** DENORMAL EXCEPTION ******************* \n");
    fpsr = 0x0009804c0270033d; /* default FPSR, with D exceptions unmasked */
    opd1 = 0x3f800000; /* 1.0 */
    fopd1 = *(float *)&opd1;
    opd2 = 0x00000001; /* smallest positive denormal */
    fopd2 = *(float *)&opd2;
    res = 0x00000000; /* 0.0 */
    fres = *(float *)&res;
    write_fpsr (fpsr);
    fres = fopd1 * fopd2;
    res = *(unsigned int *)&fres;
    printf ("after denormal exception res = %8.8x\n", res);

    /* overflow exception */
    printf ("\n******************** OVERFLOW EXCEPTION ******************* \n");
    fpsr = 0x0009804c02700337; /* default FPSR, with O exceptions unmasked */
    opd1 = 0x7f7fffff; /* 1.1...1 * 2^127 */
    fopd1 = *(float *)&opd1;
    opd2 = 0x7f7fffff; /* 1.1...1 * 2^127 */
    fopd2 = *(float *)&opd2;
    res = 0x00000000; /* 0.0 */
    fres = *(float *)&res;
    write_fpsr (fpsr);
    fres = fopd1 * fopd2;
    res = *(unsigned int *)&fres;
    printf ("after overflow exception res = %8.8x\n", res);

    /* underflow exception */
    printf ("\n******************** UNDERFLOW EXCEPTION ******************* \n");
    fpsr = 0x0009804c0270032f; /* default FPSR, with U exceptions unmasked */
    opd1 = 0x00800000; /* smallest positive normal */
    fopd1 = *(float *)&opd1;
    opd2 = 0x00800000; /* smallest positive normal */
    fopd2 = *(float *)&opd2;
    res = 0x00000000; /* 0.0 */
    fres = *(float *)&res;
    write_fpsr (fpsr);
    fres = fopd1 * fopd2;
    res = *(unsigned int *)&fres;
    printf ("after underflow exception res = %8.8x\n", res);

    /* inexact exception */
    printf ("\n******************** INEXACT EXCEPTION ******************* \n");
    fpsr = 0x0009804c0270031f; /* default FPSR, with P exceptions unmasked */
    opd1 = 0x3f800001; /* 1.0 + 1 ulp */
    fopd1 = *(float *)&opd1;
    opd2 = 0x3f800001; /* 1.0 + 1 ulp */
    fopd2 = *(float *)&opd2;
    res = 0x00000000; /* 0.0 */
    fres = *(float *)&res;
    write_fpsr (fpsr);
    fres = fopd1 * fopd2;
    res = *(unsigned int *)&fres;
    printf ("after inexact exception res = %8.8x\n", res);

}
```

The assembly language function used to unmask floating-point exceptions is:

```
write_fpsr:
  alloc r31 = ar.pfs,1,0,0,0  // r32

  // fpsr is in r32
  mov ar40 = r32;;
  br.ret.sptk b0;; // return
  .endp write_fpsr
```

3.3. Example 3. Floating-Point Exceptions Raised by Parallel Instructions from Assembly Language Code

The third example illustrates the usage of the IEEE filter for handling unmasked floating-point exceptions raised by parallel (SIMD) Itanium processor instructions. The `adjust_handler()`, `get_fp_register()`, `set_fp_register()`, and `user_handler()` functions are the same as in Examples 1 and 2, since the IEEE filter presents the exceptions individually to the user handler, one at a time, even if more than one exception per instruction occurs (so the user does not need to provide special code for handling exceptions raised by parallel instructions). Function `main()` calls this time `run_fpma()`, listed next, which is used to trigger a few of the possible exception combinations from the `fpma` instruction: invalid operation in the low half, overflow in the high half, underflow in the high half combined with denormal operand in the low half, and finally denormal operand in the high half combined with underflow in the low half. The last two cases are symmetric, to illustrate that processing of simultaneous floating-point exceptions is performed by the IEEE filter in the order low first, and high next.

```
void
main () {

  unsigned long fpsr;
  struct sigaction act, oact;
  _float128_t opd1, opd2, opd3, res;

  // Example for raising and handling unmasked floating-point exceptions from
  // parallel floating-point instructions; an assembly coded function,
  // run_fpma (), executes fpma.s0 f6 = f7, f8, f9, with pointers to the FPSR
  // and to the pairs of single precision parameters passed by the caller (the
  // single precision floating-point values are specified by their hexadecimal
  // representation, using the _float128_t data type); passing the FPSR  values
  // is necessary in order to selectively unmask (enable) floating-point
  // exceptions

  /* register user floating-point exception handler */
  act.sa_handler = (void (*)(int))adjust_handler;
  act.sa_flags = SA_NOMASK;
  sigaction (SIGFPE, &act, &oact);

  // IMPORTANT NOTE: when setting the FPSR, always leave status field 1 (sf1)
  // at its default value (traps disabled, rounding-to-nearest, 64-bit
  // precision, widest range exponent set, flush-to-zero mode not set),
  // otherwise floating-point divide, square root, remainder, and libm
  // transcendental function calculations, as well as integer divide and
  // remainder operations might generate incorrect results

  /* (no exception, invalid exception) */
  printf ("\n************** LOW HALF INVALID EXCEPTION *************** \n");
  fpsr = 0x0009804c0270033e; /* default fpsr, with I exceptions unmasked */
   opd1.W[3] = 0x0; opd1.W[2] = 0x01003e;
   opd1.W[1] = 0x3f800000; opd1.W[0] = 0x3f800000; /* (1.0, 1.0) */
   opd2.W[3] = 0x0; opd2.W[2] = 0x01003e;
   opd2.W[1] = 0x3f800000; opd2.W[0] = 0x7f800000; /* (1.0, +infinity) */
   opd3.W[3] = 0x0; opd3.W[2] = 0x01003e;
   opd3.W[1] = 0x3f800000; opd3.W[0] = 0xff800000; /* (1.0, -infinity) */
   res.W[3] = 0x0;  res.W[2] = 0x01003e;
   res.W[1] = 0x00000000;  res.W[0] = 0x00000000; /* (0.0, 0.0) */
   run_fpma (&fpsr, &res, &opd1, &opd2, &opd3);
   printf ("after low invalid exception res = %8.8x %8.8x %8.8x %8.8x\n",
        res.W[3], res.W[2], res.W[1], res.W[0]);

        /* (overflow exception, no exception) */
       printf ("\n************** HIGH HALF OVERFLOW EXCEPTION ************** \n");
       fpsr = 0x0009804c02700337; /* default fpsr, with O exceptions unmasked */
       opd1.W[3] = 0x0; opd1.W[2] = 0x01003e;
```

```
        opd1.W[1] = 0x7f7fffff; opd1.W[0] = 0x3f800000; /* (1.1...1 * 2^127, 1.0) */
        opd2.W[3] = 0x0; opd2.W[2] = 0x01003e;
        opd2.W[1] = 0x7f7fffff; opd2.W[0] = 0x3f800000; /* (1.1...1 * 2^127, 1.0) */
        opd3.W[3] = 0x0; opd3.W[2] = 0x01003e;
        opd3.W[1] = 0x3f800000; opd3.W[0] = 0x00000000; /* (1.0, 0.0) */
        res.W[3] = 0x0;  res.W[2] = 0x01003e;
        res.W[1] = 0x00000000;  res.W[0] = 0x00000000; /* (0.0, 0.0) */
        run_fpma (&fpsr, &res, &opd1, &opd2, &opd3);
        printf ("after high overflow exception res = %8.8x %8.8x %8.8x %8.8x\n",
              res.W[3], res.W[2], res.W[1], res.W[0]);

        /* (underflow exception, denormal exception) */
        printf ("\n**** HIGH HALF UNDERFLOW, LOW HALF DENORMAL EXCEPTION **** \n");
        fpsr = 0x0009804c0270032d; /* default fpsr, with U, D exceptions unmasked */
        opd1.W[3] = 0x0; opd1.W[2] = 0x01003e;
        opd1.W[1] = 0x00800000; opd1.W[0] = 0x3f800000; /* (smallest normal, 1.0) */
        opd2.W[3] = 0x0; opd2.W[2] = 0x01003e;
        opd2.W[1] = 0x00800000; opd2.W[0] = 0x00000001;
        /* (smallest normal, smallest denormal) */
        opd3.W[3] = 0x0; opd3.W[2] = 0x01003e;
        opd3.W[1] = 0x00000000; opd3.W[0] = 0x00000000; /* (0.0, 0.0) */
        res.W[3] = 0x0;  res.W[2] = 0x01003e;
        res.W[1] = 0x00000000;  res.W[0] = 0x00000000; /* (0.0, 0.0) */
        run_fpma (&fpsr, &res, &opd1, &opd2, &opd3);
        printf ("after high underflow & low denormal exception res = "
            "%8.8x %8.8x %8.8x %8.8x\n", res.W[3], res.W[2], res.W[1], res.W[0]);

       /* (denormal exception, underflow exception) */
       printf ("\n**** HIGH HALF DENORMAL, LOW HALF UNDERFLOW EXCEPTION **** \n");
       fpsr = 0x0009804c0270032d; /* default fpsr, with U, D exceptions unmasked */
       opd1.W[3] = 0x0; opd1.W[2] = 0x01003e;
       opd1.W[1] = 0x3f800000; opd1.W[0] = 0x00800000; /* (1.0, smallest normal) */
       opd2.W[3] = 0x0; opd2.W[2] = 0x01003e;
       opd2.W[1] = 0x00000001; opd2.W[0] = 0x00800000;
       /* (smallest denormal, smallest normal) */
       opd3.W[3] = 0x0; opd3.W[2] = 0x01003e;
       opd3.W[1] = 0x00000000; opd3.W[0] = 0x00000000; /* (0.0, 0.0) */
       res.W[3] = 0x0;  res.W[2] = 0x01003e;
       res.W[1] = 0x00000000;  res.W[0] = 0x00000000; /* (0.0, 0.0) */
       run_fpma (&fpsr, &res, &opd1, &opd2, &opd3);
       printf ("after high denormal & low underflow exception res = "
       "%8.8x %8.8x %8.8x %8.8x\n", res.W[3], res.W[2], res.W[1], res.W[0]);

}
```

The assembly language function used to trigger the floating-point exceptions is `run_fpma():`

```
run_fpma:
  alloc r31 = ar.pfs,5,2,0,0  // r32, r33, r34, r35, r36, r37, r38

  // &fpsr is in r32
  // &res (output) is in r33
  // &opd1 (input) is in r34
  // &opd2 (input) is in r35
  // &opd3 (input) is in r36

  mov r38 = ar40;; // save old FPSR in r38
  ld8 r37 = [r32];; // load new FPSR in r37
  mov ar40 = r37;; // set new value of FPSR
  ldf.fill f7 = [r34] // load first input argument into f7
  ldf.fill f8 = [r35] // load second input argument into f8
  ldf.fill f9 = [r36];; // load third input argument into f9
  fpma.s0 f6 = f7, f8, f9;; // f6 = f7 * f8 + f9
  mov r37 = ar40;; // store new FPSR
  st8 [r32] = r37;;
  stf.spill [r33] = f6 // store result
  mov ar40 = r38;; // restore original FPSR
  br.ret.sptk b0 // return
  .endp run_fpma
```

The output for this third example is:

```
************** LOW HALF INVALID EXCEPTION ****************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 0 0 1
USER HANDLER: enable bits PUOZI = 0 0 0 0 1
USER HANDLER: status bits PUOZI = 0 0 0 0 1
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _fp_format_fp82
USER HANDLER: operand1 value = 0000000000000ffff8000000000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _fp_format_fp82
USER HANDLER: operand2 value = 0000000000001ffff8000000000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _fp_format_fp82
USER HANDLER: operand3 value = 0000000000003ffff8000000000000000
USER HANDLER: result is not valid
USER HANDLER: invalid exception

 *** END USER HANDLER ***

after low invalid exception res = 00000000 0001003e 40000000 3f800001

************** HIGH HALF OVERFLOW EXCEPTION **************

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 1 0 0
USER HANDLER: enable bits PUOZI = 0 0 1 0 0
USER HANDLER: status bits PUOZI = 1 0 1 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _fp_format_fp82
USER HANDLER: operand1 value = 000000000001007efffff0000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _fp_format_fp82
USER HANDLER: operand2 value = 000000000001007efffff0000000000
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _fp_format_fp82
USER HANDLER: operand3 value = 0000000000000ffff8000000000000000
USER HANDLER: result is valid
USER HANDLER: result has format _fp_format_fp32
USER HANDLER: result value = 5f7ffffe
USER HANDLER: overflow exception

 *** END USER HANDLER ***

after high overflow exception res = 00000000 0001003e 3f800003 3f800000

**** HIGH HALF UNDERFLOW, LOW HALF DENORMAL EXCEPTION ****

 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 0 0 0 0
USER HANDLER: enable bits PUOZI = 0 1 0 0 0
USER HANDLER: status bits PUOZI = 0 0 0 0 0
USER HANDLER: operand1 is valid
USER HANDLER: operand1 has format _fp_format_fp82
USER HANDLER: operand1 value = 0000000000000ffff8000000000000000
USER HANDLER: operand2 is valid
USER HANDLER: operand2 has format _fp_format_fp82
USER HANDLER: operand2 value = 000000000000ff810000010000000000
```

```
USER HANDLER: operand3 is valid
USER HANDLER: operand3 has format _fp_format_fp82
USER HANDLER: operand3 value = 00000000000000000000000000000000
USER HANDLER: result is not valid
USER HANDLER: denormal exception

 *** END USER HANDLER ***


 *** BEGIN USER HANDLER ***

USER HANDLER: round to nearest
USER HANDLER: double extended precision
USER HANDLER: single floating-point multiply-add operation
USER HANDLER: cause bits PUOZI = 0 1 0 0 0
USER HANDLER: enable bits PUOZI = 0 1 0 0 0
USER HANDLER: status bits PUOZI = 0 1 0 0 0
```

4. Authors

Marius Cornea (marius.cornea@intel.com)

5. Acknowledgments

Bob Norin and Asit Mallick had a significant contribution in creating this document.

6. References

[1] ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, IEEE, New York, 1985.

[2] Intel Corporation, *The Itanium™ Architecture Software Developer's Manual*, at http://developer.intel.com/design/ia-64/manuals

[3] Cornea-Hasegan, M. and Norin, B., *IA-64 Floating-Point Operations and the IEEE Standard for Binary Floating-Point Arithmetic*, Intel Technology Journal, Q4, 1999, at http://developer.intel.com/technology/itj/q41999.htm

[4] Intel Corporation, *Itanium™ Processor Floating-point Software Assistance and Floating-point Exception Handling*, at http://developer.intel.com/software/products/opensource/libraries/num.htm